Secondo: An Extensible Database System for Research and Teaching

Ralf Hartmut Güting

Fernuniversität Hagen, Germany

Secondo - Overview

An environment for implementing DBMS with new kinds of data models, suitable for research prototyping and teaching. Developed in the last ten years or so at University of Hagen, Germany.

- no fixed data model
- system frame can be filled with implementations of different data models, e.g.
  - relational
  - object-relational
  - graph/network-oriented
  - sequence-oriented
- goes beyond extensibility just by attribute data types
- system frame contains data model independent parts of a DBMS
- data model dependent parts implemented in algebra modules
- current “contents”: basically a relational system with several advanced data type packages

Open source software, available at

http://dna.fernuni-hagen.de/Secondo.html/

Secondo - Overview

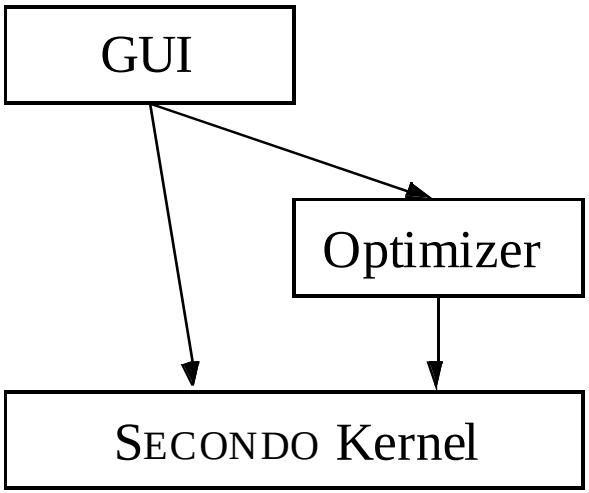
Three major components:

- Secondo Kernel:
  - implements specific data models
  - extensible by algebra modules
  - provides query processing over the implemented algebras
  - implemented on top of BerkeleyDB storage manager
  - written in C++
- Optimizer:
  - core capability: conjunctive query optimization
  - currently supports a relational model with an SQL-like language
  - written in PROLOG
- GUI:
  - extensible interface for an extensible DBMS like Secondo
  - extensible by viewers
  - sophisticated spatial / spatio-temporal viewer, extensible by data types
  - written in Java

Secondo - Overview

Components work together:

- GUI sends executable query (query plan) to the kernel, displays result
- GUI sends query to optimizer, receives plan, sends plan to kernel, displays result
- optimizer sends commands and executable queries to kernel to get information about DB objects, e.g. selectivities



Secondo - Kernel

– Secondo Kernel:

- implements specific data models
- extensible by algebra modules
- provides query processing over the implemented algebras
- implemented on top of BerkeleyDB storage manager
- written in C++

Secondo - Kernel

Underlying Concept: Second-Order Signature

A formalism to describe

- a *descriptive algebra*, defining a data model and query language,
- an *executable algebra*, specifying a collection of data structures and operations capable of representing the data model and implementing the query language,
- rules to enable a query optimizer to map descriptive algebra terms to executable algebra terms (*query plans*).

Secondo - Kernel - Second-Order Signature

Basic idea: Use two coupled signatures. The first signature describes a type system, the second an algebra over the types generated by the first signature.

Example:

	$\rightarrow$	DATA	*int*, *real*, *bool*
DATA	$\rightarrow$	SET	*set*

Terms of the first signature (= types of the type system)

int, *real*, *bool*, *set*(*int*), *set*(*real*), *set*(*bool*)

are sorts of the second.

$\forall$ *data* in DATA.

data $\times$ *data*	$\rightarrow$	*bool*	$=, \neq, <, \leq, \geq, >$

Secondo - Kernel - Second-Order Signature

Specifying a Descriptive Algebra

kinds IDENT, DATA, TUPLE, REL

type constructors

	$\rightarrow$	DATA	*int*, *real*, *bool*, *string*
(IDENT $\times$ DATA)$^+$	$\rightarrow$	TUPLE	*tuple*
TUPLE	$\rightarrow$	REL	*rel*

Example term (= type, schema):

rel(*tuple*([(name, *string*), (age, *int*)])

Secondo - Kernel - Second-Order Signature

Specifying a Descriptive Algebra

operators

$\forall$ *data* in DATA.

data $\times$ *data* $\rightarrow$ *bool* $=, \neq, <, \leq, \geq, >$ _ # _

$\forall$ *rel*: *rel*(*tuple*) in REL.

rel $\times$ (*tuple* $\rightarrow$ *bool*) $\rightarrow$ *rel* **select** _ # [_]

$\forall$ *tuple*: *tuple*(*list*) in TUPLE, *attrname* in IDENT,

member(*attrname*, *attrtype*, *list*).

tuple $\times$ *attrname* $\rightarrow$ *attrtype* **attr** # (_, _)

A query:

people **select**[**fun** (p: person) **attr**(p, age) > 20]

Secondo - Kernel - Second-Order Signature

Specifying an Executable Algebra

kinds IDENT, DATA, TUPLE, RELREP

type constructors

		DATA	*int*, *real*, *bool*, *string*
(IDENT × DATA)$^+$	→	TUPLE	*tuple*
TUPLE	→	RELREP	*srel*, *relrep*

Secondo - Kernel - Second-Order Signature

Specifying an Executable Algebra

operators

$\forall$ ...

tuple in TUPLE.

relrep(*tuple*)	$\rightarrow$	*stream*(*tuple*)	**feed**	_ #
stream(*tuple*) $\times$ (*tuple* $\rightarrow$ *bool*)	$\rightarrow$	*stream*(*tuple*)	**filter**	_ # [_]
stream(*tuple*)	$\rightarrow$	*srel*(*tuple*)	**consume**	_ #

A query plan:

people **feed** **filter**[**fun** (p: person) **attr**(p, age) > 20] **consume**

Secondo - Kernel - Second-Order Signature

Commands

A database is a pair (*T*, *O*) where *T* is a finite set of named types, and *O* a finite set of named objects. Seven basic commands to manipulate a database:

```
type <identifier> = <type expression>
delete type <identifier>

create <identifier>: <type expression>
update <identifier> := <value expression>
let <identifier> = <value expression>
delete <identifier>

query <value expression>
```

Secondo - Kernel - Commands

Basic Commands	Inquiries

`type <identifier> = <type expression>`
`delete type <identifier>`
`create <identifier>: <type expression>`
`update <identifier> := <value expression>`
`let <identifier> = <value expression>`
`delete <identifier>`
`query <value expression>`
	`list type constructors`
	`list operators`
	`list algebras`
	`list algebra <identifier>`
	`list databases`
	`list types`
	`list objects`

Databases	**Transactions**

`create database <identifier>`
`delete database <identifier>`
`open database <identifier>`
`close database`
	`begin transaction`
	`commit transaction`
	`abort transaction`

Import and Export	

`save database to <file>`
`restore database <identifier> from <file>`
`save <identifier> to <file>`
`restore <identifier> from <file>`	

Secondo - Kernel

Demo: The Kernel

```
SecondoTTYBDB
list databases
list algebras
list algebra RTreeAlgebra
open database opt
query 3 * 5
create x: int
update x := 7
delete inc
let inc = fun(n:int) n + 1
query inc(inc(7))
query Orte
query plz count
query plz feed filter[.Ort = “Hagen”] consume
query plz_Ort plz exactmatch[“Hagen”] consume
query Orte feed {o} plz feed {p} hashjoin[Ort_o, Ort_p, 99997]
  count
```

Secondo - Kernel

Demo: The Kernel

Next example uses objects:

```
Kreis: rel(tuple([KName: string, ..., Gebiet: region]))
kreis_Gebiet: rtree(tuple([KName: string, ..., Gebiet: region]))
magdeburg: region
```

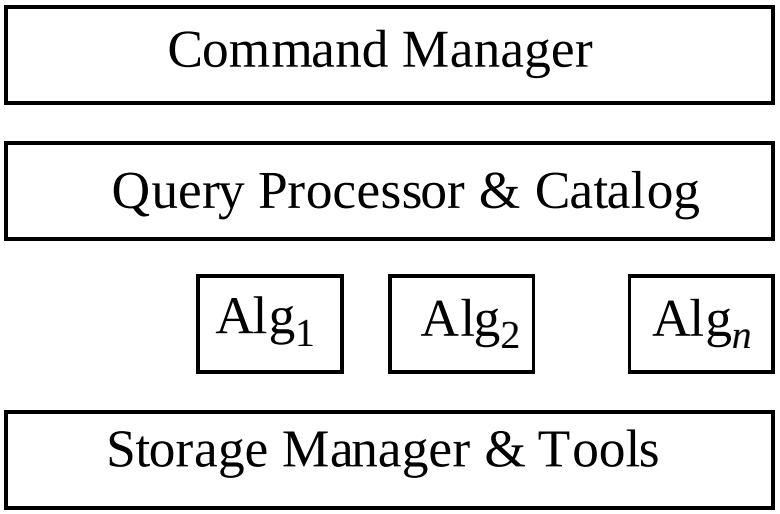
The following query finds neighbour counties of **magdeburg**:

```
query
  kreis_Gebiet Kreis windowintersects[bbox(magdeburg)]
  filter[.Gebiet adjacent magdeburg]
  filter[not(.KName contains "Magdeburg")]
  project[KName]
  consume
```

```
rtree(Tuple) x rel(Tuple) x rectangle   -> stream(Tuple)    windowintersects   #[ _ ]
region                                  -> rectangle        bbox               #( _ )
region x region                         -> bool             adjacent           # _
```

Secondo - Kernel

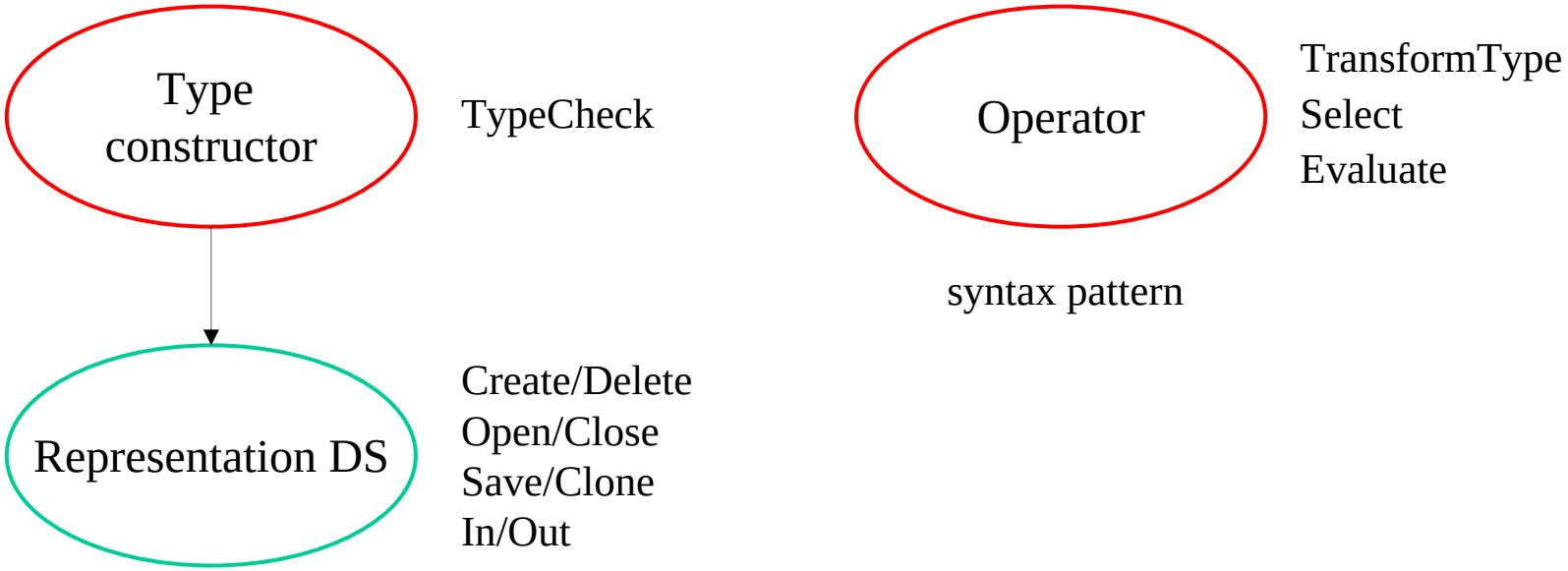
Rough architecture:



Secondo - Kernel

Structure of Algebra Modules

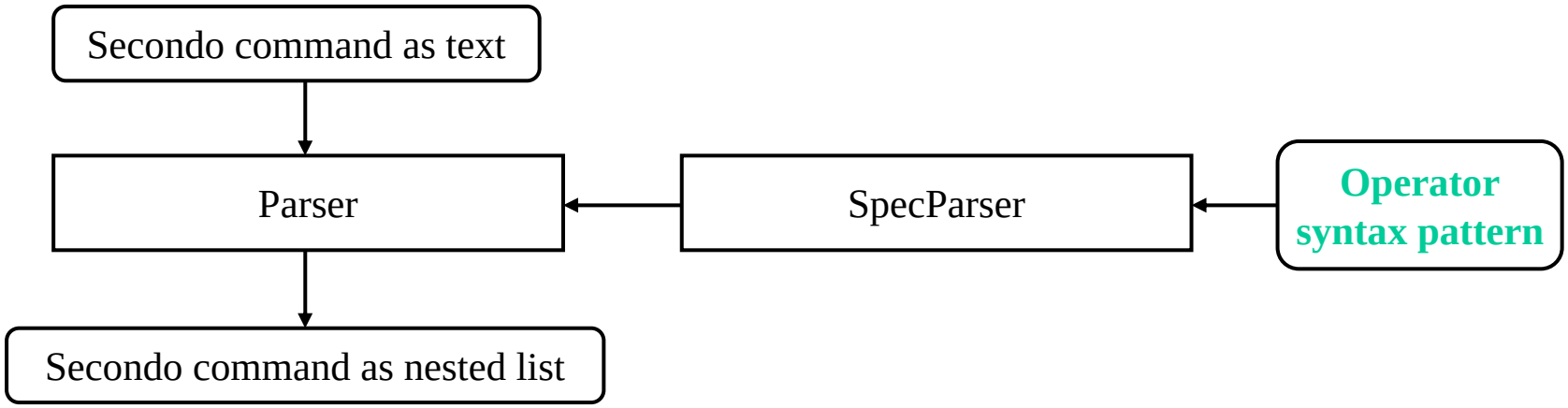
Each algebra module offers some type constructors and some operators. The module contains:



Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

```
query plz feed filter[.Ort = "Hagen"] consume
```




```
(query (consume (filter (feed plz) (fun (tuple1 TUPLE) (= (attr tuple1
Ort) "Hagen")))))
```

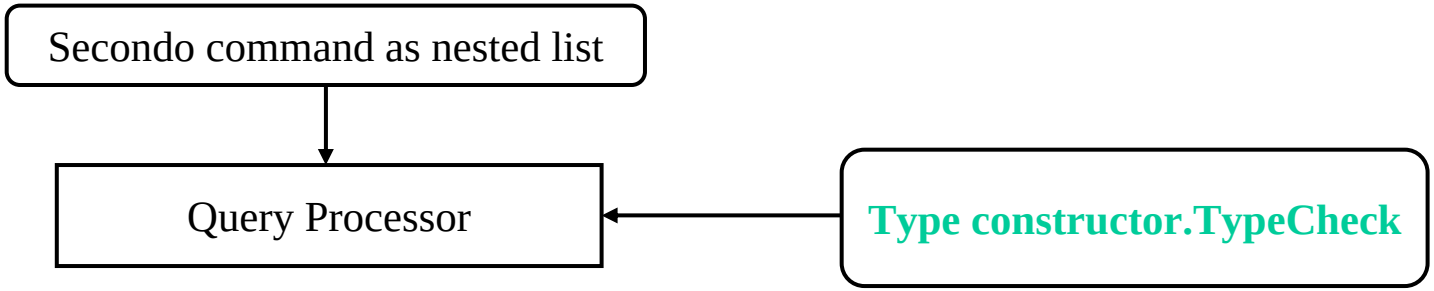
```
operator feed alias FEED pattern _ op
operator attr alias ATTR pattern op (_, _)
```

Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

Processing Type Expressions

```
type <identifier> = <type expression>
create <identifier>: <type expression>
```

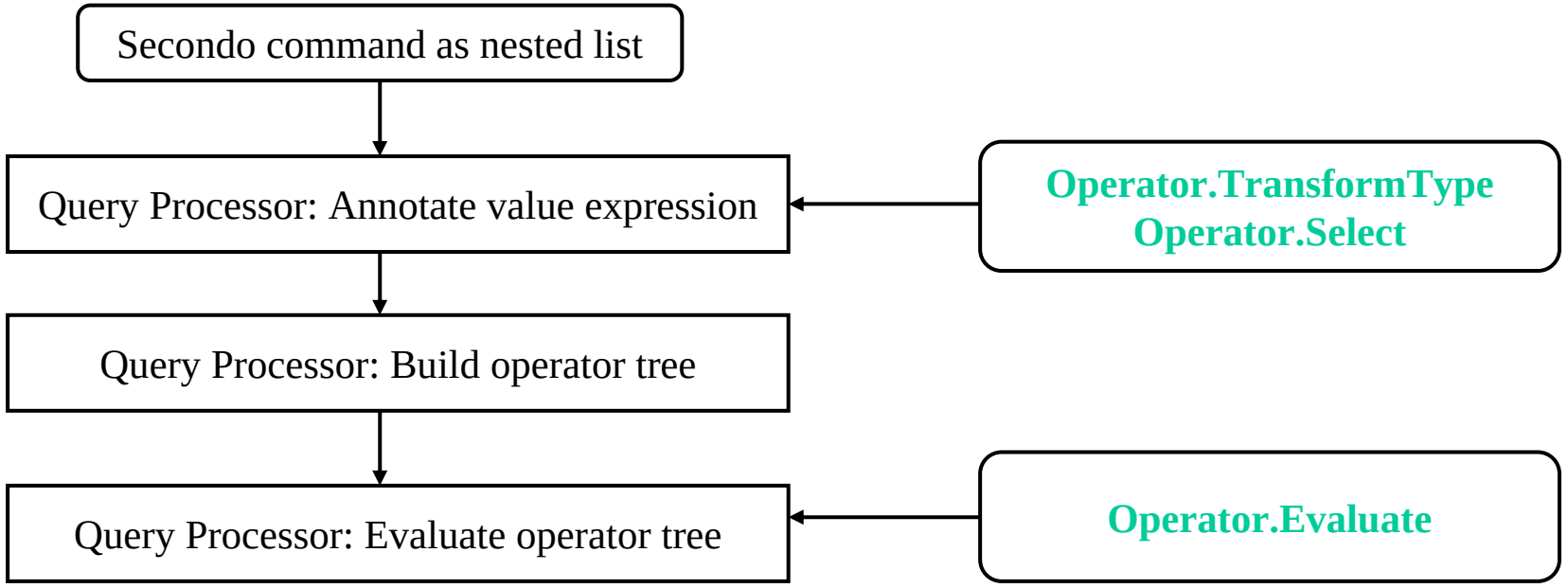


Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

Processing Value Expressions

```
update <identifier> := <value expression>
let <identifier> = <value expression>
query <value expression>
```



The Secondo Kernel – Query Processing

Representation of Types, Queries, Values

Nested lists are used extensively in the system, to represent:

- type expressions
- value expressions (queries, query plans)
- constants in queries
- values of data types in the *external representation* (for exchange with other applications)

Type expressions:

rel(*tuple*([(name, *string*), (pop, *int*), (country, *string*)])

```
(rel (tuple ((name string) (pop int) (country string))))
```

Queries:

cities **feed** **filter**[**fun** (c: city) **attr**(c, pop) > 1000000] **consume**

```
(consume (filter (feed cities) (fun (c city) (> (attr c pop)
  1000000))))
```

The Secondo Kernel – Query Processing – Representation of Types, Queries, Values

Values in external representation (e.g. list representation of a polygon):

```
((1.0 3.8) (4.0 3.8) (2.5 6.0))
```

Generic constants in queries: (<type expression> <value list>)

```
(polygon ((1.0 3.8) (4.0 3.8) (2.5 6.0)))
```

Values in internal representation: a single word of storage.

The Secondo Kernel – Query Processing

Processing Type Expressions: Kind Checking

Process commands

type <identifier> = <type expression>

create <identifier>: <type expression>

Check whether type constructors are applied correctly in the type expression.

```
(type city_rel =
  (rel (tuple ((name string) (pop int) (country string))))
)
```

		DATA	*int*, *real*, *bool*, *string*
(IDENT × DATA)$^{+}$	→	TUPLE	*tuple*
TUPLE	→	REL	*rel*

Use *TypeCheck* function of each type constructor. System frame provides a function *CheckKind* for each kind that can be used to check quantification over kinds.

The Secondo Kernel – Query Processing

Processing Value Expressions: Type Checking and Evaluation

Process commands

update <identifier> := <value expression>

let <identifier> = <value expression>

query <value expression>

Three steps:

– Annotating the Query

– Building the Operator Tree

– Evaluation

The Secondo Kernel – Query Processing – Processing Value Expressions

Annotating the Query

```
(consume (filter (feed cities) (fun (c city) (> (attr c pop)
 1000000))))
```

Process nested list tree bottom-up, annotating each node with its type and other information. Includes type checking.

For each atom or sublist *s*, return a list of the form

```
((s class ...) type)
```

e.g.

```
((feed operator 2 1) none)
```

When an application of an operator to a list of arguments

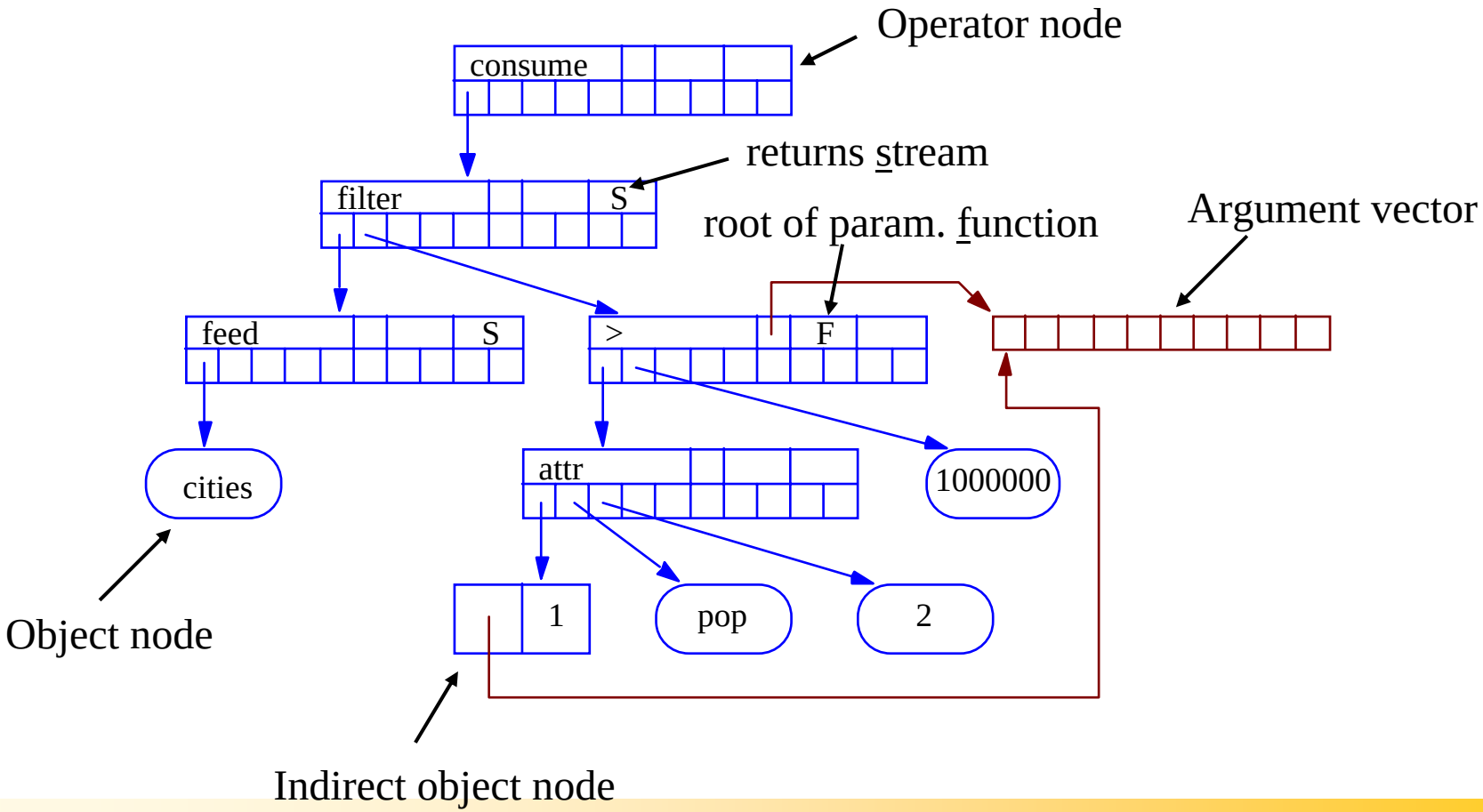
```
(op arg1 ... argn)
```

is recognized, the operator's *TransformType* function is called with the types of arg_1, ..., arg_n to check whether argument types are correct and to return the result type.

The Secondo Kernel – Query Processing – Processing Value Expressions

Building the Operator Tree

```
(consume (filter (feed cities) (fun (c city) (> (attr c pop)
 1000000))))
```



The Secondo Kernel – Query Processing – Processing Value Expressions

Evaluation

function *eval* (*t*: node): WORD
input: a node *t* of the operator tree
output: the value of the subtree rooted in *t*
method:
  if *t* is an object or indirect object **then** lookup the value and return it
  else {*t* is an operator node}
    for each subtree t_i of *t* **do**
      if the root of t_i is marked as function (F) or stream (S)
      then $arg_i := t_i$
      else $arg_i := eval(t_i)$
      end
    end;
    call the operator's evaluation function with argument vector *arg* and return its result
  end
end *eval*;

Secondo - Kernel

Cooperation Between Query Processor and Algebra Modules

Operator evaluation functions all have the same generic interface. They call query processor primitives to evaluate arguments (subtrees) that are functions or streams:

- getArguments
- request

to evaluate a parameter function subtree

- open
- request
- close
- received

to communicate with argument streams

Evaluation functions for stream operators return special values YIELD or CANCEL.

Secondo - Optimizer

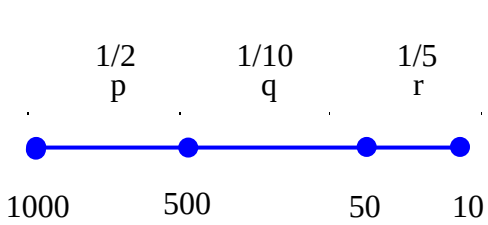
– Optimizer:

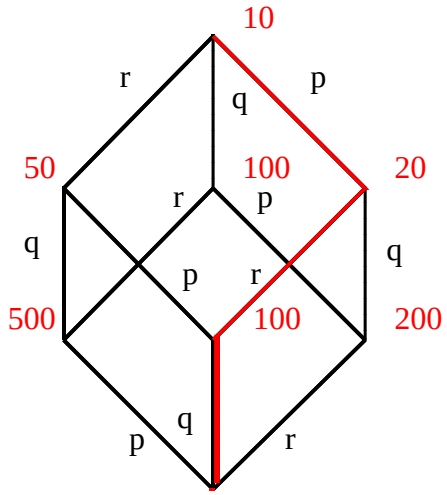
- core capability: conjunctive query optimization
- currently supports a relational model with an SQL-like language
- written in PROLOG

Secondo - Optimizer

Performs conjunctive query optimization: given a set of relations and a set of selection or join predicates, find a good plan. Uses a new algorithm for this.

Based on **shortest path search** in a **predicate order graph**.





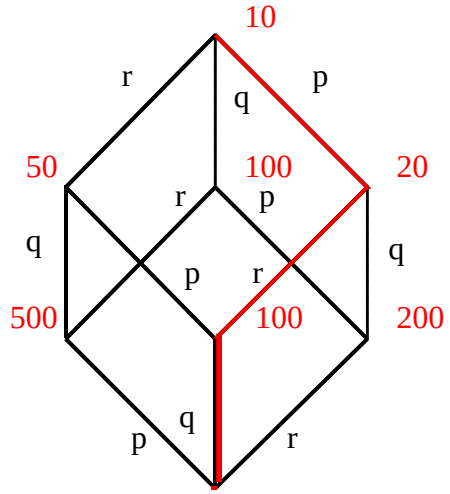
Selectivities of predicates are determined in advance by evaluating selections and joins on small sample relations. Selectivities once determined are stored for future use.

Optimizer implements an SQL-like language in a notation adapted to PROLOG.

Secondo - Optimizer

Optimization algorithm:

1. For given relations and predicates, construct the POG.
2. For each edge, construct plan edges. Controlled by optimization rules for selections and joins.
3. For given sizes of arguments and selectivities of predicates, assign sizes to all nodes (intermediate results). Also annotate edges of POG with selectivities.
4. For each plan edge, compute its cost. Based on sizes of arguments, selectivity along the edge, and cost function for each operator occurring in a plan edge.
5. Use algorithm of Dijkstra to find a shortest path from bottom to top node through the graph of plan edges. This is the plan.



Secondo - Optimizer

Demo: The Optimizer

```
open 'database opt'.    [updateRel(orte), updateRel(plz), assert(noProgress).]

sql select count(*) from [orte as o, plz as p] where [o:bevt < 500, o:ort =
    p:ort]
```

Step 1: construct the POG

```
writeNodes.
writeEdges.
```

Step 2: construct plan edges

```
writePlanEdges.
```

Step 3: assign sizes

```
writeSizes.
```

Step 4: compute cost edges

```
writeCostEdges
```

Step 5: compute shortest path

```
dijkstra(0, 3, Path, Cost), plan(Path, Plan), plan_to_atom(Plan, Query).
```

Secondo - Optimizer

Demo: The Optimizer

A more complex query:

```
sql select count(*)
from [orte as o, plz as p1, plz as p2, plz as p3]
where [
   o:ort = p1:ort,
   p2:plz = p1:plz + 7,
   (p2:plz mod 5) = 0,
   p2:plz > 30000,
   o:ort contains "o",
   o:bevt > 200,
   o:bevt < 700,
   p3:plz = p2:plz + 40]

writeSizes.
```

Secondo - Optimizer

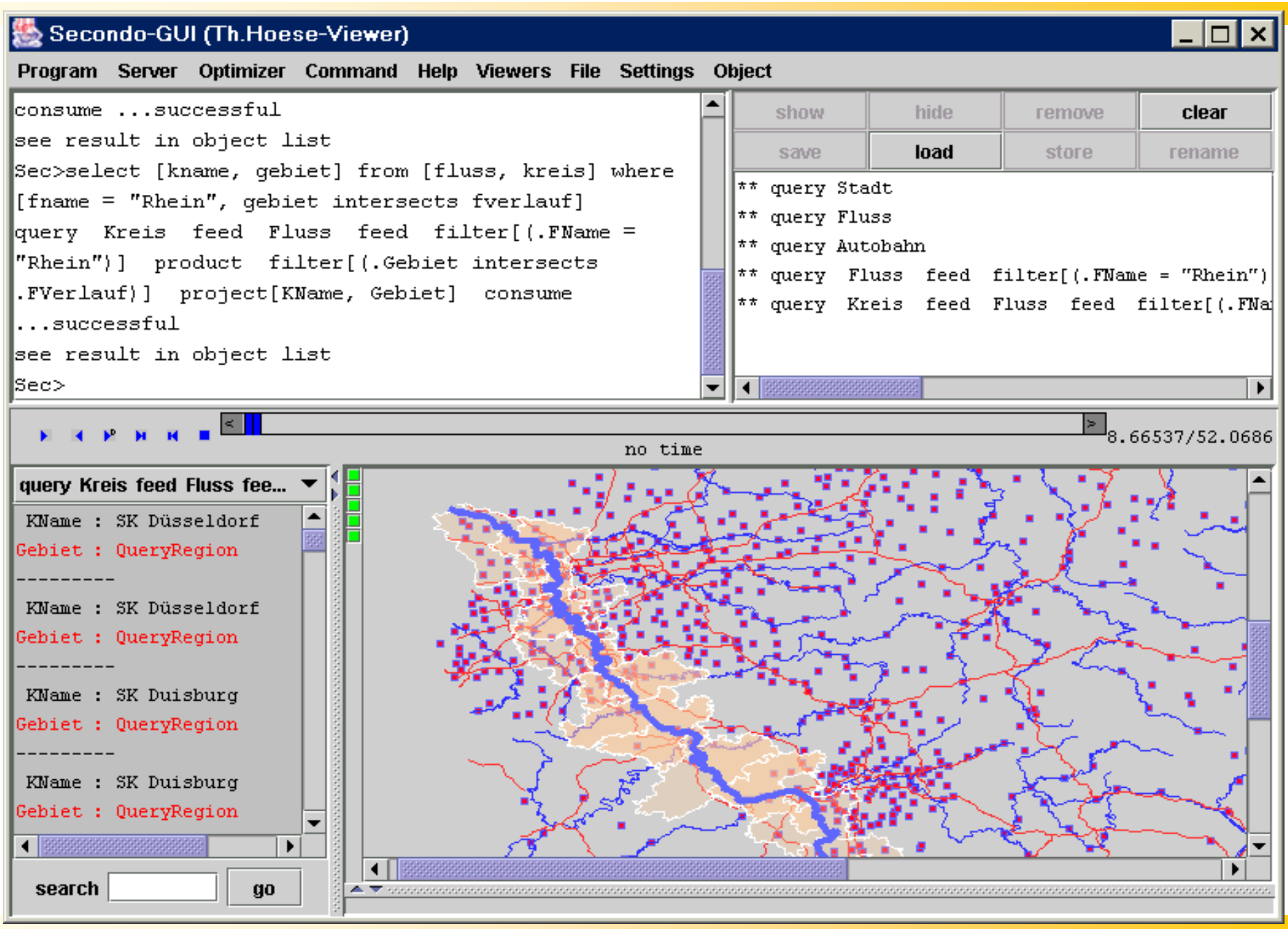
Some Interesting Properties

- Simple concept, easy to understand, relatively easy to implement
- Guaranteed to find the optimal plan
  [among available plans, assuming correct cost functions, selectivity estimates, attribute independence]
- Exponential complexity, POG has 2^n nodes, n * 2^{n-1} edges. Works fine for up to about 10 predicates (less than a second).
- If an efficient plan exists, Dijkstra explores only a small part of the POG
- A variant builds only the part of the POG that is explored by Dijkstra.
- Deals with expensive predicates (important for non-standard applications such as moving objects)
- Selectivity estimation by sampling works automatically as soon as a new operation is implemented (histograms not feasible)
- Sample queries for selection predicates cheap
- Sample queries for join predicates more expensive, but exhausted after a while.
- Easy to write optimization rules in PROLOG

Secondo - Graphical User Interface

– GUI:

- extensible interface for an extensible DBMS like Secondo
- extensible by viewers
- sophisticated spatial / spatio-temporal viewer, extensible by data types
- written in Java


Secondo-GUI (Th.Hoese-Viewer)
Program Server Optimizer Command Help Viewers File Settings Object
consume ...successful
see result in object list
Sec>select [kname, gebiet] from [fluss, kreis] where
[fname = "Rhein", gebiet intersects fverlauf]
query Kreis feed Fluss feed filter[(.FName =
"Rhein")] product filter[(.Gebiet intersects
.FVerlauf)] project[KName, Gebiet] consume
...successful
see result in object list
Sec>
show
hide
remove
clear
save
load
store
rename
** query Stadt
** query Fluss
** query Autobahn
** query Fluss feed filter[(.FName = "Rhein")
** query Kreis feed Fluss feed filter[(.FNa
no time
8.66537/52.0686
query Kreis feed Fluss fee...
KName : SK Düsseldorf
Gebiet : QueryRegion
KName : SK Düsseldorf
Gebiet : QueryRegion
KName : SK Duisburg
Gebiet : QueryRegion
KName : SK Duisburg
Gebiet : QueryRegion
search
go

A Moving Objects DBMS Prototype

Demo: GUI and Moving Objects

```
[start Javagui]

open database berlintest
query UBahn
query train7
query trajectory(train7)
query deftime(train7)
query train7 atinstant six30
query val(train7 atinstant sixthirty)
query theminute(2003,11,20,6,25)
query train7 atperiods
  theperiod(theminute(2003,11,20,6,25), theminute(2003,11,20,6,39)
query thecenter
query train7 at thecenter
query deftime(train7 at thecenter)
```

A Moving Objects DBMS Prototype

Demo: GUI and Moving Objects

```
query Trains count
query Trains feed filter[.Trip present eight] consume

[start optimizer server]

query mehringdamm
select * from trains
  where [trip present eight, trip passes mehringdamm]

select [val(trip atinstant eight) as ateight] from trains
  where [trip present eight, trip passes mehringdamm]
```

A Moving Objects DBMS Prototype

Example: Create a time dependent density animation

```
observe x-extension 30000, y-extension 20000
create a 6 x 4 raster of squares of size 5000
lower left corner at (-4000, 1000)

let r1 = [const rect value (-4000.0 1000.0 1000.0 6000.0)]
query r1
query ten
query r1 feed transformstream consume

query r1 feed transformstream
  ten feed filter[.no < 7] {t1}
  ten feed filter[.no < 5] {t2}
  product product
  projectextend[; Field:
      .elem translate[(.no_t1 – 1) * 5000.0, (.no_t1 – 1) * 5000.0]]
  consume
```